Litrepl: Literate Paper Processor Promoting Transparency More Than Reproducibility

Sergei Mironov
sergei.v.mironov@proton.me

January 18, 2025

Abstract

Litrepl is a lightweight text processing tool designed to recognize and evaluate code sections within Markdown or LaTeX documents. This functionality is useful for both batch document section evaluation and interactive coding within a text editor, provided a straightforward integration is established. Inspired by Project Jupyter, Litrepl aims to facilitate the creation of research documents. In the light of recent developments in software deployment, however, we have shifted our focus from informal reproducibility to enhancing transparency in communication with programming language interpreters, by either eliminating or clearly exposing mutable states within the communication process.

1 Statement of Need

The concept of *Literate Programming* was formulated by Donald Knuth, suggesting a shift in focus from writing code to explaining to human beings what we want a computer to do. This approach is embodied in the WEB system[6] and its descendant family of tools, whose name refers to a text document format containing the "network" of code sections interleaved with text.

The system could both render such text into human-readable documentation and compile machine-executable code. Over time, this concept has evolved, showing a trend towards simplification (Ramsey [10]).

Concurrently, a concept of human-computer interaction often called the *Read-Evaluate-Print Loop* or "REPL" gained traction, notably within the LISP an APL communities (Spence [12], McCarthy [7], Iverson [4]).

The combination of a command-line interface and a language interpreter enables incremental and interactive programming, allowing users to directly modify the interpreter state. By maintaining human involvement in the loop, this approach is believed to facilitate human thought processes (Granger and Pérez [2]).

A significant milestone in this field was the IPython interpreter (Perez and Granger [8]), which later evolved into the Jupyter Project. Its creators introduced a new document format called the Jupyter Notebook (Kluyver et al. [5]), characterized by a series of logical sections of various types, including text and code, which could directly interact with programming language interpreters. This interactive communication, akin to REPL style programming, allows the creation of well-structured documents suitable for presentations and sharing. The concept underpinning these developments is termed *Literate Com-*

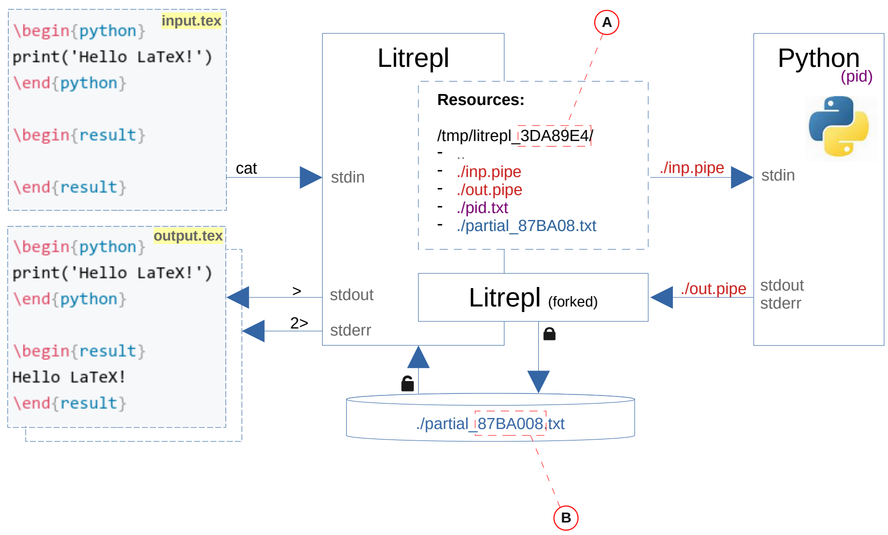

Figure 1: Litrepl resource allocation diagram. Hash **A** is computed based on the Litrepl working directory and the interpreter class. Hash **B** is computed based on the contents of the code section.

puting[9], which includes goals of spanning a wide audience range, boosting reproducibility, and fostering collaboration. To achieve these objectives, several technical decisions were made, notably the introduction of bidirectional communication between the computational core, known as the Jupyter Kernel, and the Notebook web-based renderer, along with another layer of client-server communication between the web-server and the user's web browser.

While we recognize the importance of all goals within the Literate Computing framework, we think that the goal of reproducibility is more important than others. Addressing it alone would suffice to enhance communication among time-separated and space-separated researchers and significantly expand the audience. However, as it became clear (Dolstra, Löh, and Pierron [1]), this challenge extends beyond the scope of a single human-computer interaction system, and even beyond the typical boundaries of software distribution management for a particular programming language. A comprehensive solution to the software deployment problem operates at the entire operating system level.

Following Vallet, Michonneau, and Tournier [13], we suggest changing the focus of human-computer interaction towards simplicity and transparency. We saw an opportunity to implement a tool that would offer REPL-style editing, be compatible with existing code editors and text formats, thus maintaining familiar editing practices, contain only a few hidden state variables, and have a significantly smaller codebase.

We introduce *Litrepl*[1], a text processor that employs the following approaches: first, utilizing straightforward bidirectional text

[1] `https://github.com/sergei-mironov/litrepl`

streams for inter-process communication with language interpreters to evaluate code; second, advocating for the reuse of existing text document formats. In both the Markdown and LaTeX evaluators we have implemented, simplified parsers are used to distinguish code and result sections from the rest of the document. As of now, we support Python and Shell interpreter families, as well as a custom large language model communication interpreter. Finally, we strive to leverage POSIX[3] operating system facilities as much as possible.

2 How it works

Litrepl is implemented as a command-line text utility. Its primary function is to take a text document as input, process it according to specified command-line arguments and environment settings, and then output the resulting document through its standard output.

The operation of Litrepl is best illustrated through the example below. Consider the document named `input.tex`:

```
$ cat input.tex
\begin{python}
import sys
print(f"I use {sys.platform} btw!")
\end{python}
\begin{result}
\end{result}
```

This document contains a Python code section and an empty result section marked with the corresponding LaTeX environment tags. To "execute" the document we pipe it though the Litrepl processor as follows:

```
$ cat input.tex | litrepl
\begin{python}
import sys
print(f"I use {sys.platform} btw!")
\end{python}
\begin{result}
I use linux btw!
\end{result}
```

Now we can see the expected statement about the author's operating system. The side-effect of this execution is the started session of the python interpreter which is now running in the background. We can modify its state by adding more section to the document and executing them selectively or e.g. by accessing `litrepl repl python` terminal. So for example, setting `sys.platform` to another value and re-evaluating the document would yield a different statement.

2.1 Interfacing Interpreters

Litrepl communicates with interpreters using two uni-directional text streams: one for writing input and another for reading outputs. To establish effective communication, the interpreter should conform to the following general assumptions:

- Synchronous single-user mode, which is implemented in most interpreters.
- A capability to disable command line prompts. Litrepl relies on the echo response, as described below, rather than on prompt detection.
- The presence of an echo command or equivalent. The interpreter must be able to echo an argument string provided by the user in response to the echo command.

In Litrepl, these details are hardcoded for several prominent interpreter families, which we refer to as *interpreter classes*. At the time of writing, Litrepl supports three such classes: `python`, `sh`, and `ai`. Using these names in command line arguments, users can configure how to map code section labels to the correct class and specify which interpreter command to execute to start a session for each class. For example, to select a Bourne-Again Shell interpreter as `sh`, we add the `--sh-interpreter=/usr/bin/bash` argument assuming that this binary is present in the system.

2.2 Session Management

Litrepl's ability to maintain interpreter sessions in the background is crucial for enabling a Read-Eval-Print Loop (REPL) environment. The associated resources, shown in Figure 1, are stored as files within an auxiliary directory.

If not specified by command-line arguments or environment variables, the directory path is automatically derived from the interpreter class name, the current working directory, and the OS's temporary file location.

The auxiliary directory includes two POSIX pipes for interpreter I/O and a file recording the running interpreter's process ID, aiding session management.

When a code section is evaluated, Litrepl assigns a response file name derived from hashing the code. This response file stores the output from the interpreter.

During evaluation, Litrepl spawns a response reader process with a soft lock, active until the interpreter completes and responds to an echo probe. The state machine that operates the probe is the only added hidden state in the entire system.

If the response exceeds the configured duration, Litrepl outputs a partial result tag, which is recognized and reevaluated in subsequent runs. Figure 2 shows an example partial result section.

Litrepl provides **start**, **stop**, **restart** and **status** commands to control background sessions, so, for example

```
$ litrepl restart python
```

stops the Python interpreter if it was running and starts a new instance of it. The **interrupt** command sends an interruption signal to the interpreter. Finally, the **repl** command establishes direct communication with the interpreter, allowing manual inspection of its state. The **help** command prints the detailed description of each command and the configuration arguments available.

2.3 Parsing and Evaluation

Litrepl abstracts documents as a straightforward sequence comprising code, result, and text sections. Additionally, Litrepl identifies ignore blocks, which act as comments that prevent enclosed sections from being evaluated.

Template grammars similar to the illustrative example in Figure 3 are encoded for Markdown and LaTeX formats. Before each run, Litrepl calls Lark (Shinan [11]) to compile a customized parser and uses it to access the sections.

Evaluation results are written back into the result sections, and the entire document is printed. At this stage, certain conditions can be optionally checked. First, setting `--pending-exitcode` to a non-zero value instructs Litrepl to report an error if a section takes longer than the timeout to evaluate. Second, setting `--exception-exitcode` directs Litrepl to detect Python exceptions. Lastly, `--irreproducible-exitcode` triggers an error if the evaluation result doesn't match the text initially present in the result section.

The last option implements the only formal check for aiding reproducibility that Litrepl provides.

```
\begin{result}
... some output ...
[LR:/tmp/litrepl/python/partial_c335adc.txt]
\end{result}
```

Figure 2: Example partial result section that ends with a continuation tag, where `LR:` is a distinguishable prefix, and the rest is the filename storing a response of the interpreter to the section being evaluated.

```
document       ::= (code | result | ignore | text)*
code           ::= (code-normal | code-comment)
result         ::= (result-normal | result-comment)
code-normal    ::= "\begin{MARKER}" text "\end{MARKER}"
code-comment   ::= "% MARKER" text "% noMARKER"
result-normal  ::= "\begin{result}" text "\end{result}"
result-comment ::= "% result" text "% noresult"
ignore         ::= "% ignore" text "% noignore"
text           ::= ...
```

Figure 3: An illustrative grammar template for LaTeX documents where marker serves as a parameter configured via command-line arguments for each supported interpreter class.

3 Discussion

The technical decision to abstract interpreters using text streams comes with both advantages and disadvantages. A key advantage is simplicity. However, there are notable negative aspects. First, there is no parallel evaluation at the communication level, meaning the interpreter is locked until it completes the evaluation of one snippet before proceeding to the next. Second, the transferable data type is restricted to text-only streams.

We argue that the lack of parallel execution at the communication level can be mitigated using interpreter-specific parallelism, where supported. For instance, Python programs can utilize various subprocess utilities, while shell programs have full access to shell job control.

The restriction to text-only data types presents a more fundamental limitation. Litrepl lifts this restriction by supporting text-only document formats. Both LaTeX and Markdown incorporate non-text data without encoding it directly, instead relying on references and side channels, such as the file system or network resource identifiers. Consequently, Litrepl shares, for example, the benefits of human-readable representation in version control systems and the penalties, such as the need to explicitly organize side-channel data transfer.

Another controversial technical decision is transferring the entire document as input and output, which can negatively impact performance. Our experience shows that the system performs adequately for documents of a few thousand lines. However, larger documents may experience uncomfortable delays, even on modern computers. Despite this, we choose to maintain this interface because it simplifies editor integration. A typical plugin can pipe the whole document through the tool using just a few lines of code.

A more performance-oriented integrations can make pre-parsing and pipe only relevant document parts. For these approaches, Litrepl offers the `print-regexp` command, which outputs the anchor regexp in several common formats.

4 Conclusion

The tool is implemented in Python in under 2K lines of code according to the LOC metric, and has only two Python dependencies so far, at the cost of the dependency on the POSIX operating system interfaces. Needless to say, we used Litrepl to evaluate and verify the examples presented in this document.

References

[1] Eelco Dolstra, Andres Löh, and Nicolas Pierron. "NixOS: A purely functional Linux distribution". In: *Journal of Functional Programming* 20.5–6 (2010), pp. 577–615. DOI: 10.1017/S0956796810000195. URL: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/nixos-a-purely-functional-linux-distribution/C1ACBA2A51D2E5466820F5B5086EA2CE.

[2] Brian E. Granger and Fernando Pérez. "Jupyter: Thinking and Storytelling With Code and Data". In: *Computing in Science & Engineering* 23.2 (2021), pp. 7–14. DOI: 10.1109/MCSE.2021.3059263. URL: https://ieeexplore.ieee.org/document/9387490.

[3] IEEE and The Open Group. *POSIX Standard: Base Specifications, Issue 8*. Standard Issue 8. Adopted Standard. The Open Group, June 2024, p. 4107. URL: https://publications.opengroup.org/c243.

[4] Kennet E. Iverson. *A Programming Language*. USA: John Wiley & Sons, Inc., 1962. ISBN: 0471430145. URL: https://archive.org/embed/aprogramminglanguage1962.

[5] Thomas Kluyver et al. "Jupyter Notebooks - a publishing format for reproducible computational workflows". In: *International Conference on Electronic Publishing*. 2016. URL: https://api.semanticscholar.org/CorpusID:36928206.

[6] Donald Ervin Knuth. "Literate Programming". In: *Computer/law journal*. 1984. URL: https://api.semanticscholar.org/CorpusID:1200693.

[7] John McCarthy. "Recursive Functions of Symbolic Expressions and their Computation by Machine". In: 1959. URL: https://api.semanticscholar.org/CorpusID:267886520.

[8] Fernando Perez and Brian E. Granger. "IPython: A System for Interactive Scientific Computing". In: *Computing in Science and Engg.* 9.3 (May 2007), pp. 21–29. ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.53. URL: https://doi.org/10.1109/MCSE.2007.53.

[9] Fernando Pérez and Brian E. Granger. *Project Jupyter: Computational Narratives as the Engine of Collaborative Data Science*. Tech. rep. 2015. URL: https://blog.jupyter.org/project-jupyter-computational-narratives-as-the-engine-of-collaborative-data-science-2b5fb94c3c58.

[10] Norman Ramsey. "Literate Programming Simplified". In: *IEEE Softw.* 11.5 (Sept. 1994), pp. 97–105. ISSN: 0740-7459. DOI: 10.1109/52.311070. URL:

https://doi.org/10.1109/52.311070.

[11] Erez Shinan. *Lark*. URL: https://github.com/lark-parser/lark.

[12] Robert Spence. *APL Demonstration 1975*. 2020. URL: https://www.youtube.com/watch?v=_DTpQ4Kk2wA.

[13] Nicolas Vallet, David Michonneau, and Simon Tournier. "Toward practical transparent verifiable and long-term reproducible research using Guix". In: *Scientific Data* 9.1 (Oct. 2022), p. 597. ISSN: 2052-4463. DOI: 10.1038/s41597-022-01720-9. URL: https://doi.org/10.1038/s41597-022-01720-9.